Visualisation de Système de fonctions itérées sur GPU

Nathan Jacquinet

1er juillet 2024

Résumé

Implémentation d'une méthode de visualisation de système de fonction itérée (IFS) sur GPU en annulant l'utilisation de mémoire à l'aide d'un codage arithmétique. code source

1 Introduction

Pour démarrer le sujet, j'ai d'abord développé un backend Vulkan, premièrement obtenu sur ce tutorial en ligne : Vulkan Tutorial, puis adapté. Vulkan est une API graphique de bas niveau, elle nous permet, en échange de temps de développement, d'avoir un accès très granulaire et flexible aux GPU. Un des avantages à cela, c'est que l'on peut envoyer, à l'aide de PushConstant, une faible quantité (minimum 128 octets suivant les GPU) de données de manière extrêmement efficace, comparé à un Uniform.

2 Implémentation sur CPU

Une première implémentation sur CPU a été faite de sorte à mieux appréhender l'algorithme et à commencer à rencontrer les contraintes. Par exemple, nous ne pouvons pas faire d'allocation de mémoire plus grande que 4 Go en une seule fois sur un GPU. (Le problème est observable sur OpenGL, ainsi que sur Vulkan, probablement aussi sur DirectX). Nous rencontrons ce problème rapidement, par exemple lors de l'allocation du tampon nécessaire pour accueillir tous les points calculés d'un IFS. Une solution serait de multiplier les allocations, mais il faudrait alors gérer les frontières des tampons lors de l'exécution de l'IFS.

Ou alors, il faut d'abord faire une allocation classique, exécuter l'IFS, puis copier en fragmentant si nécessaire par paquet de 4 Go dans des tampons accessibles par le GPU.

Du point de vue de l'algorithme, un IFS peut se représenter sous forme d'arbre, où chaque nœud représente une itération n, et chaque branche représente une application d'une transformation de l'IFS.

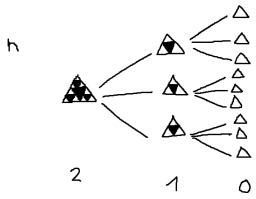

Figure 1 – Exemple arbre IFS pour le triangle de Sierpiński

Nous pouvons observer que les branches ont certaines parties similaires, plus précisément, que pour chaque itération, nous avons n-1 sous-arbre redondant (avec n, le nombre de transformations de l'IFS). Autrement dit, que nous devons effectuer plusieurs fois la même suite de calculs. Il pourrait donc être intéressant de limiter les calculs superflus et de réutiliser les sous-arbres similaires.

L'implémentation sur CPU effectue cela :

```
int main() {
   //Transform are 4D matrices bc of heterogeneous coord
    void IFS(glm::vec3* object, size_t nbVertices, std::vector<glm::mat4>&
        trans, uint32_t nbIteration)
    {
        //For each iteration
        for (int i = 0; i < nbIteration; ++i)
        {

            //For each transform, we write result from the end to the start of
                the buffer
            for (size_t j = 0; j < trans.size(); ++j)
            {
                //We apply tranform for each point
                for (size_t k = 0; k < nbVertices; ++k)
                {
                    object[((trans.size() - 1 - j)) * nbVertices + k] = glm::
                        vec4(object[k], 1.0) * trans[j];
                }
            }
            //Update size of the total object
            nbVertices = nbVertices * trans.size();
        }
    }
}
```

Listing 1 – Algorithme itératif exécutant un IFS en ne calculant qu'un seul sous-arbre par itération.

L'algorithme est plutôt intéressant, une amélioration possible serait de paralléliser la partie dans laquelle nous appliquons les transformations aux points. En effet, c'est la seule partie où il n'y a pas de dépendances séquentielles ! Mais attention à bien calculer la dernière transformation en dernière (car chaque multiplication utilise le résultat de l'IFS d'avant, qui est stocké en début de tampon, et la dernière itération écrasera celle-ci).

Cette approche pourrait être intéressante lorsque l'on a beaucoup de transformations différentes dans l'IFS.

Mais le problème majeur, comme mentionné plus haut, est le stockage des points.
Par exemple, pour un IFS générant le triangle de Sierpiński, qui possède alors trois transformations par itération, avec un simple triangle en tant que forme de départ, nous avons besoin de

$$s = b^n * 4 * 3$$

octets. Avec n le nombre d'itérations et b le nombre de points de la géométrie initiale, et 4 étant le nombre d'octets que prend une valeur flottante, et trois le nombre de points initiaux.

3 Implémentation sur GPU

Une autre solution serait de trouver un moyen de réduire l'utilisation de mémoire, par exemple en évitant de stocker le résultat de l'IFS, mais de plutôt le calculer et les afficher directement sans les stocker.
Pour cela, nous allons directement calculer chaque branche de l'arbre directement depuis le Vertex shader, à l'aide d'instanciation et d'un codage arithmétique de chacune de ces branches.

Voici l'approche :
— Nous avons toujours notre élément de départ, que nous stockons dans un tampon de Vertex.
— Les différentes transformations de l'IFS seront écrites dans un Uniform.

— Les informations comme le nombre d'instances, le nombre d'itérations, le nombre de transformations seront intégrées à la commande de dessin, grâce au PushConstant, puis envoyées au GPU.

Du côté du shader, un segment entre 0 et 1 est subdivisé uniformément par le nombre de transformations différentes, ensuite chaque branche de l'arbre sera codée par une valeur flottante entre 0 et 1.

Pour chaque invocation du vertex shader, une valeur flottante sera alors calculée en fonction de leur indice d'instance et du nombre d'instances invoquées, qui représente ainsi le codage arithmétique de la branche à effectuer pour une instance donnée.

```
int main() {
    float currentValue = float(gl_InstanceIndex) / float(PushConstants.
        maxInstance - 1);
    float f_nbTransform = float(PushConstants.nbTransformation);
    float segmentSize = float(1.0f) / f_nbTransform;
```

Listing 2 – Calcule de la valeur local associé à chaque instance

Ensuite, en fonction de quel sous-segment la valeur est, nous effectuons la transformation associée à ce sous-segment.

```
    //For each iteration
    for (int i = 0; i < nbIteration; ++i)
    {
        //Get the indice of the transfom to apply
        float f_no = floor(currentValue * f_nbTransform);
        uint no = min(uint(f_no), PushConstants.nbTransformation-1);
        model = model * transpose(transforms.t[no]);
```

Listing 3 – Calcule de l'indice de la transformation associé à la valeur

On réadapte la valeur entre 0 et 1, par rapport au sous-segment où il était, et on répète alors ce processus le nombre d'itérations demandé.

```
        //Remap the value between 0 and 1 for the next iteration
        float lower = float(no) * segmentSize;
        currentValue = (currentValue - lower) / segmentSize;
    }
```

Listing 4 – Mise à l'échelle de la valeur

Ainsi, nous supprimons l'empreinte mémoire, mais cela est remplacé par le fait d'effectuer énormément de fois les mêmes calculs. Cependant, l'architecture d'un GPU est parfaitement adaptée pour ce genre d'utilisation. Nous pouvons observer une très bonne amélioration de vitesse de rendu, et nous pouvons aussi aller encore plus loin dans les itérations, car nous n'avons plus la contrainte de mémoire.

Néanmoins, la limitation de cette approche est le nombre maximal d'instances, qui est sur Vulkan quantifié sur un entier non signé de 32 bits, soit un peu plus de quatre milliards. Une solution, beaucoup plus simple que celle pour le problème d'emprunter mémoire, serait de faire plusieurs DrawCall, en ajoutant dans le PushConstant les bornes des valeurs à générer pour le codage des branches par DrawCall. Cependant, le temps de rendu commence à être long lorsque l'on est proche de ce maximum.

3.1 Analyse des performances

Nous allons comparer les performances en temps d'exécution de l'algorithme et aussi en empreinte mémoire.
J'ai choisi l'IFS du triangle de Sierpiński, qui possède trois transformations. Et mon objet de départ est un triangle.

TABLE 1 – Comparaison des algorithmes en termes de temps d'exécution et d'empreinte mémoire

Nombre d'itérations	**CPU**		**GPU**		**Speed up**	
	Temps	**Mémoire**	**Temps**	**Mémoire**	**Temps**	**Mémoire**
3	76800 ns	972 B	< 0.1 ms	0 B	N/A	Inf
6	2,08 ms	26,24 kB	< 0.1 ms	0 B	N/A	Inf
9	19.27 ms	708,59 kB	0.2 ms	0 B	96 350	Inf
12	521,90 ms	19,14 MB	0.6 ms	0 B	870	Inf
15	13,86 s	516,56 MB	16.8 ms	0 B	825	Inf
16	42,51 s	1,55 Go	50.2 ms	0 B	846	Inf
17	N/A	N/A	151.4 ms	0 B	N/A	Inf
18	N/A	N/A	454.6 ms	0 B	N/A	Inf
19	N/A	N/A	1.367 s	0 B	N/A	Inf
20	N/A	N/A	4.116 s	0 B	N/A	Inf

Notes : les résultats du benchmark ont été obtenus avec un Intel Core I7 13700K (@5.4Ghz) et une Nvidia GTX 1080.

La durée d'exécution de la version CPU est calculée sur le temps d'exécution de l'algorithme uniquement, elle ne prend pas en compte le transfert ainsi que l'affichage de la première image du résultat. (La mesure est alors sous-estimé). De plus, à partir de 17 itérations, nous avons besoin de plus de quatre Go de mémoire, c'est pour cela qu'il n'y a pas de données.

Pour ce qui est de l'implémentation sur GPU, la mesure se fait en mesurant le temps de rendu d'une image, puisque l'algorithme s'exécute directement dans le vertex shader. Cela implique qu'il y a aussi le temps d'exécution des fragments shader. (la mesure est alors surestimé).

Vulkan nous permet à l'aide d'extensions de mesurer de façon très précise le temps d'exécution des commandes qu'on envoie au GPU. Il serait idéal d'implémenter cela plus tard.

3.2 Conclusion

L'implémentation sur GPU est intéressante principalement par ses performances. Une version permettant d'écrire dans un tampon serait utile à implémenter et permettrait de solidifier la comparaison des deux algorithmes, même si le résultat devrait être similaire.

Nous voyons aussi que malgré la redondance de certains calculs, l'implémentation sur GPU est beaucoups plus rapide. Et qu'avec le codage arithmétique, cela retire la contrainte de mémoire, rendant l'algorithme infiniment plus efficace en mémoire, en plus de pouvoir aller encore plus loin dans les itérations.